PAGE No.
DATE

1. A perception is linear but XOR is not linearly separable.
MLPs add hidden layers and nonlinearity, enabling multiple decision boundaries.

2. Becoz the composition of linear function is still linear.
Gradients shrink becoz of ~~multiple~~ repeated multiplication of derivatives < 1 like sigmoid or tanh.
ReLU doesn't saturate for positive inputs & has derivative 1.

3. Necessary becoz self attention lacks sequence order.
Sinusoidal PE is fixed & extrapolates to longer sequences.
Absolute PE is flexible but limited to trained sequence
ROPE encodes relative position in attention space

4. Query - what a Token is searching for
Key - what token represents.
Value - info returned after attention.

Scaling by $\sqrt{d_k}$ prevents large dot products that cause softmax saturation.
- Becoz tokens strongly attend to themselves.

5. Multiple heads allow the model to attend to different features & relationships simultaneously.

d_model - total embedding dimension
h - no. of attention heads

d-head = dmodel / h

6- Greedy decoding selects highest probability token at each step
Beam keeps multiple candidate sequences.
Eg - Greedy → I am fine
Beam → I am finally feeling much better

1. $d\_head = \frac{dmodel}{h} = \frac{768}{12} = 64$
Q, K, V has a weight matrix of size 768 x 768
parameters per matrix = $768^2 = 589824$
Total for Q, K, V = 3 x 589 824.

2. $Softmax = \frac{e^{x_i}}{\sum e^{x_i}}$
$\longrightarrow e^2 + e^1 + e^0 = 11.11$

$\frac{e^2}{11.11} = 0.665$ $\frac{e^1}{11.11} = 0.245$ $\frac{e^0}{11.11} = 0.090$

[0.665, 0.245, 0.090]

3. Numpy —

```
import numpy as np
def attention (Q, K, V):
    dK = Q.shape [-1]
    Scores = Q@K.T/np.sqrt (dk)
    weights = np.exp (scores) / np.sum(np.exp(scores), axis=1, keepdims=True)
    output = weights@V
    return output, weights
```

4. Numpy Version – Masked Attention.

```
import numpy as np
def masked_attention (Q, K, V):
    dk = Q.shape [-1]
    scores = Q @ K.T / np.sqrt (dK)

    seq_len = scores.shape[0]
    mask = np.triu(np.ones((seq_len, seq_len)), k=1)

    scores = np.where (mask==1, -np.inf, scores)

    weights = np.exp(scores)/np.sum (np.exp(Scores), axis=1,
    output = weights@V                                keepdims = True)
    return output, weights.
```